CC410: System Programming

Dr. Manal Helal – Fall 2014 – Lecture 3

Learning Objectives

- Study More SIC Programming Examples
- Understand CISC Machines
- Understand RISC Machines

1.3.3 SIC Programming Examples

```
        LDA     ZERO        INITIALIZE INDEX VALUE TO 0
        STA     INDEX
ADDLP   LDX     INDEX       LOAD INDEX VALUE INTO REGISTER X
        LDA     ALPHA,X     LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X      ADD WORD FROM BETA
        STA     GAMMA,X     STORE THE RESULT IN A WORD IN GAMMA
        LDA     INDEX       ADD 3 TO INDEX VALUE
        ADD     THREE
        STA     INDEX
        COMP    K300        COMPARE NEW INDEX VALUE TO 300
        JLT     ADDLP       LOOP IF INDEX IS LESS THAN 300
        .
        .
        .
INDEX   RESW    1           ONE-WORD VARIABLE FOR INDEX VALUE
.                           ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
.                           ONE-WORD CONSTANTS
ZERO    WORD    0
K300    WORD    300
```

1.3.3 SIC Programming Examples

```
        LDS     #3          INITIALIZE REGISTER S TO 3
        LDT     #300        INITIALIZE REGISTER T TO 300
        LDX     #0          INITIALIZE INDEX REGISTER TO 0
ADDLP   LDA     ALPHA,X     LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X      ADD WORD FROM BETA
        STA     GAMMA,X     STORE THE RESULT IN A WORD IN GAMMA
        ADDR    S,X         ADD 3 TO INDEX VALUE
        COMPR   X,T         COMPARE NEW INDEX VALUE TO 300
        JLT     ADDLP       LOOP IF INDEX VALUE IS LESS THAN 300
        .
        .
        .
.                           ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
```

(b)

Figure 1.5 Sample indexing and looping operations for (a) SIC and (b) SIC/XE.

1.3.3 SIC Programming Examples

```
INLOOP    TD       INDEV        TEST INPUT DEVICE
          JEQ      INLOOP       LOOP UNTIL DEVICE IS READY
          RD       INDEV        READ ONE BYTE INTO REGISTER A
          STCH     DATA         STORE BYTE THAT WAS READ
          .
          .
          .
OUTLP     TD       OUTDEV       TEST OUTPUT DEVICE
          JEQ      OUTLP        LOOP UNTIL DEVICE IS READY
          LDCH     DATA         LOAD DATA BYTE INTO REGISTER A
          WD       OUTDEV       WRITE ONE BYTE TO OUTPUT DEVICE
          .
          .
          .
INDEV     BYTE     X'F1'        INPUT DEVICE NUMBER
OUTDEV    BYTE     X'05'        OUTPUT DEVICE NUMBER
DATA      RESB     1            ONE-BYTE VARIABLE
```

Figure 1.6 Sample input and output operations for SIC.

1.3.3 SIC Programming Examples

```
            JSUB    READ            CALL READ SUBROUTINE
            .
            .
            .
.                                   SUBROUTINE TO READ 100-BYTE RECORD
READ        LDX     ZERO            INITIALIZE INDEX REGISTER TO 0
RLOOP       TD      INDEV           TEST INPUT DEVICE
            JEQ     RLOOP           LOOP IF DEVICE IS BUSY
            RD      INDEV           READ ONE BYTE INTO REGISTER A
            STCH    RECORD,X        STORE DATA BYTE INTO RECORD
            TIX     K100            ADD 1 TO INDEX AND COMPARE TO 100
            JLT     RLOOP           LOOP IF INDEX IS LESS THAN 100
            RSUB                    EXIT FROM SUBROUTINE
            .
            .
            .
INDEV       BYTE    X'F1'           INPUT DEVICE NUMBER
RECORD      RESB    100             100-BYTE BUFFER FOR INPUT RECORD
.                                   ONE-WORD CONSTANTS
ZERO        WORD    0
K100        WORD    100
```

1.3.3 SIC Programming Examples

```
            JSUB    READ            CALL READ SUBROUTINE
            .
            .
            .
.                                   SUBROUTINE TO READ 100-BYTE RECORD
READ        LDX     #0              INITIALIZE INDEX REGISTER TO 0
            LDT     #100            INITIALIZE REGISTER T TO 100
RLOOP       TD      INDEV           TEST INPUT DEVICE
            JEQ     RLOOP           LOOP IF DEVICE IS BUSY
            RD      INDEV           READ ONE BYTE INTO REGISTER A
            STCH    RECORD,X        STORE DATA BYTE INTO RECORD
            TIXR    T               ADD 1 TO INDEX AND COMPARE TO 100
            JLT     RLOOP           LOOP IF INDEX IS LESS THAN 100
            RSUB                    EXIT FROM SUBROUTINE
            .
            .
            .
INDEV       BYTE    X'F1'           INPUT DEVICE NUMBER
RECORD      RESB    100             100-BYTE BUFFER FOR INPUT RECORD
```

- CISC & RISC Machines
  - Chapters 1.4 & 1.5 of Leland Beck's "*System Software*" book.

Traditional (CISC) Machines

- Complex Instruction Set Computers (CISC)
  - Complicated instruction set
  - Different instruction formats and lengths
  - Many different addressing modes
  - e.g. VAX or PDP-11 from DEC
  - e.g. Intel x86 family (**more details in another presentation on moodle**)

Pentium Pro Architecture (1/5)

- Memory
  - Physical level: *byte addresses*, *word*, *doubleword*
  - Logical level: *segments* and offsets
  - In some cases, a segment can also be divided into *pages*
  - The segment/offset address specified by the programmer is translated into a physical address by the x86 *MMU* (*Memory Management Unit*)

Pentium Pro Architecture (2/5)

- Registers
  - General-purpose registers:
    - EAX, EBX, ECX, EDX: data manipulation
    - ESI, EDI, EBP, ESP: address
  - Special-purpose registers:
    - EIP: next instruction
    - FLAGS: status word
    - CS: code segment register
    - SS: stack segment register
    - DS, ES, FS, and GS: data segments
  - Floating-point unit (FPU)
  - Registers reserved for system programs

16-bit segment registers

Pentium Pro Architecture (3/5)

- Data Formats
  - Integers:
    - 8-, 16-, 32-bit binary numbers
    - Negative values: 2's complement
    - FPU can also handle 64-bit signed integers
    - The least significant part of a numeric value is stored at the lowest-numbered address *(little-endian)*
    - Binary coded decimal (BCD)
      - unpacked: 0000____0000____0000____.....0000____
      - packed: |____|____|____|____|____|____|.....|____|____|
  - Floating-point data formats
    - *Single-precision*: 32 bits=24+7-bit exponent+sign bit
    - *Double-precision*: 64 bits=53+10-bit exponent+sign bit
    - *Extended-precision*: 80 bits=64+15-bit exponent+sign bit

Pentium Pro Architecture (4/5)

- Instruction Formats
  - *Prefix* (optional) containing flags that modify the operation of instruction
    - specify repetition count, segment register, etc.
  - *Opcode* (1 or 2 bytes)
  - *Operands and addressing modes*

- Addressing Modes
  - *TA=(base register)+(index register)*(scale factor)+displacement*
  - *Base register*: any general-purpose registers
  - *Index register*: any general-purpose registers except ESP
  - *Scale factor*: 1, 2, 4, 8
  - *Displacement*: 8-, 16-, 32- bit value
  - Eight addressing modes

Pentium Pro Architecture (5/5)

- Instruction Set
  - 400 different machine instructions
    - R-to-R instructions, R-to-M instructions, M-to-M instructions
    - immediate values,
  - Special purpose instructions for high-level programming language
    - entering and leaving procedures,
    - checking subscript values against the bounds of an array
- Input and Output
  - Input is performed by instructions that transfer one byte, word, or doubleword from an I/O port to register EAX, output is performed similarly.
  - *Repetition prefixes* allow these instructions to transfer an entire string in a single operation

RISC Machines

- RISC system
  - Instruction
    - Standard, fixed instruction format
    - Single-cycle execution of most instructions
    - Memory access is available only for load and store instruction
    - Other instructions are register-to-register operations
    - A small number of machine instructions, and instruction format
  - A large number of general-purpose registers
  - A small number of addressing modes

RISC Machines

- Three RISC machines
  - SPARC family
  - PowerPC family
  - Cray T3E

UltraSPARC (1/8)

- Sun Microsystems (1995)
- SPARC stands for scalable processor architecture
- SPARC, SuperSPARC, UltraSPARC are upward compatible and share basic structure:
  - Memory
  - Registers
  - Data formats
  - Instruction Formats

UltraSPARC (2/8)

- Byte addresses
  - Two consecutive bytes form halfword
  - Four bytes form a word
  - Eight bytes form doubleword
- Alignment
  - Halfword are stored in memory beginning at byte address that are multiples of 2
  - Words begin at addresses that are multiples of 4
  - Doublewords at addresses that are multiples of 8
- Virtual address space
  - UltraSPARC programs can be written using 2^{64} bytes virtual memory divided into pages in physical memory or disk.
  - Memory Management Unit handles address translation and loading

UltraSPARC (3/8)

- Registers
  - ~100 general-purpose registers
  - Any procedure can access only 32 registers (r0~r31)
    - First 8 registers (r0~r8) are global, i.e. they can be access by all procedures on the system (r0 is zero)
    - Other 24 registers can be visualised as a window through which part of the register file can be seen
  - Program counter (PC)
    - The address of the next instruction to be executed
  - Condition code registers
  - Other control registers

UltraSPARC (4/8)

- Data Formats
  - Integers are 8-, 16-, 32-, 64-bit binary numbers
  - 2's complement is used for negative values
  - Support both big-endian and little-endian byte orderings
    - (big-endian means the most significant part of a numeric value is stored at the lowest-numbered address)
  - Three different floating-point data formats
    - Single-precision, 32 bits long (23 + 8 + 1)
    - Double-precision, 64 bits long (52 + 11 + 1)
    - Quad-precision, 78 bits long (63 + 16 + 1)

UltraSPARC (5/8)

- Three Instruction Formats
  - 32 bits long
  - The first 2 bits identify which format is being used
  - Format 1: call instruction
  - Format 2: branch instructions
  - Format 3: remaining instructions

UltraSPARC (6/8)

- Addressing Modes
  - Immediate mode
  - Register direct mode
  - Memory addressing

Mode	Target address calculation
PC-relative*	TA= (PC)+displacement {30 bits, signed}
Register indirect with displacement	TA= (register)+displacement {13 bits, signed}
Register indirect indexed	TA= (register-1)+(register-2)

*PC-relative is used only for branch instructions

UltraSPARC (7/8)

- Instruction Set
  - <100 instructions
  - Pipelined execution
    - While one instruction is being executed, the next one is fetched from memory and decoded
  - Delayed branches
    - The instruction immediately following the branch instruction is actually executed before the branch is taken
  - Special-purpose instructions
    - High-bandwidth block load and store operations
    - Special “atomic” instructions to support multi-processor system

UltraSPARC (8/8)

- Input and Output
  - A range of memory locations is logically replaced by device registers
  - Each I/O device has a unique address, or set of addresses
  - No special I/O instructions are needed

PowerPC Architecture (1/8)

- POWER stands for Performance Optimisation with Enhanced RISC
- History
  - IBM (1990) introduced POWER in 1990 with RS/6000
  - IBM, Apple, and Motorola formed an alliance to develop PowerPC in 1991
  - The first products were delivered near the end of 1993
  - Recent implementations include PowerPC 601, 603, 604

PowerPC Architecture (2/8)

- Memory
  - Halfword, word, doubleword, quadword (16 bytes)
  - May instructions may execute more efficiently if operands are aligned at a starting address that is a multiple of their length
  - Virtual space 2^{64} bytes
  - Fixed-length segments, 256 MB
  - Fixed-length pages, 4KB
  - MMU: virtual address -> physical address

PowerPC Architecture (3/8)

- Registers
  - 32 general-purpose registers, GPR0~GPR31, 32 or 64 bit long
  - FPU, containing 32 - 64 bit FP registers
  - Condition code register reflects the result of certain operations, and can be used as a mechanism for testing and branching
  - Link Register (LR) and Count Register (CR) are used by some branch instructions
  - Machine Status Register (MSR)

PowerPC Architecture (4/8)

- Data Formats
  - Integers are 8-, 16-, 32-, 64-bit binary numbers
  - 2's complement is used for negative values
  - Support both big-endian (default) and little-endian byte orderings
  - Two different floating-point data formats
    - single-precision, 32 bits long (23 + 8 + 1)
    - double-precision, 64 bits long (52 + 11 + 1)
  - Characters are stored using 8-bit ASCII codes

PowerPC Architecture (5/8)

- Seven Instruction Formats
  - 32 bits long
  - The first 6 bits identify specify the opcode
  - Some instruction have an additional extended opcode
  - The complexity is greater than SPARC
  - Fixed-length makes decoding faster and simple than VAX and x86

PowerPC Architecture (6/8)

- Addressing Modes
  - Immediate mode, register direct mode
  - Memory addressing

Mode	Target address calculation
Register indirect	TA=(register)
Register indirect with indexed	TA=(register-1)+(register-2)
Register indirect with immediate indexed	TA=(register)+displacement {16 bits, signed}

  - Branch instruction
  -

Mode	Target address calculation
Absolute	TA= actual address
Relative	TA= current instruction address + displacement {25 bits, signed}
Link Register	TA= (LR)
Count Register	TA= (CR)

PowerPC Architecture (7/8)

- Instruction Set
  - 200 machine instructions
    - More complex than most RISC machines
    - e.g. floating-point “multiply and add” instructions that take three input operands
    - e.g. load and store instructions may automatically update the index register to contain the just-computed target address
  - Pipelined execution
    - More sophisticated than SPARC
  - Branch prediction

PowerPC Architecture (8/8)

- Input and Output
  - Two different modes
    - Direct-store segment: map virtual address space to an external address space
    - Normal virtual memory access

CISC vs. RISC

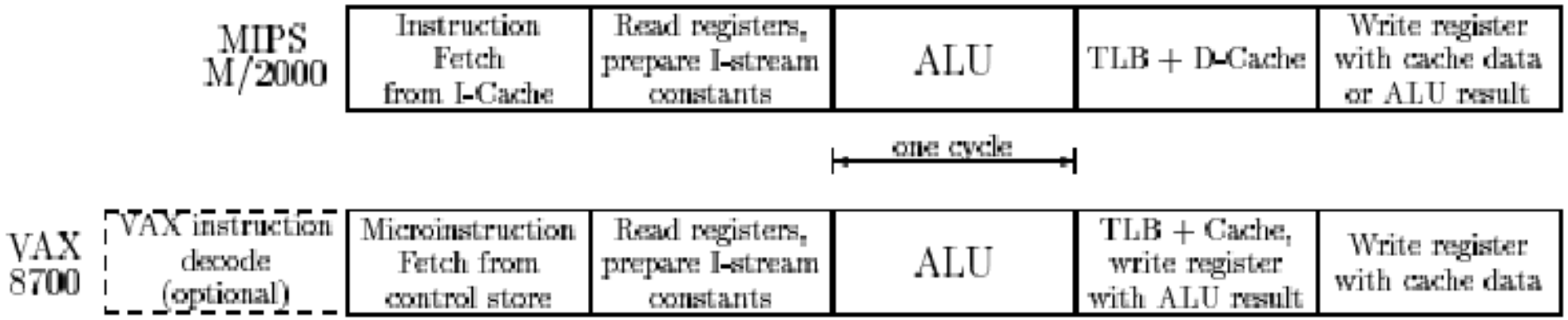
- MIPS M/2000(from RISC) and VAX 8700(from CISC)

  -same underlying organization

- Most recent compilers were used for each of the two machines
- Cycle time determined through machine independent features but its same
- Spec 1 Release benchmarks used

MIPS M/2000(from RISC) and VAX 8700(from CISC)

CPU Pipeline Abstractions MIPS and VAX



Simplified illustration of the two instruction pipelines. A new instruction (microinstruction on the VAX) can start every cycle. The VAX decode cycle is omitted when a microroutine is more than one microinstruction long

- MIPS instruction fetch stage matches with VAX micro- instruction fetch stage
  - Large set of general purpose registers
  - Single cycle instructions
  - Delayed branches

About MIPS and VAX

- Strong organisational similarities
  - Ex: CPU Pipeline abstractions match up closely
- VAX Microinstruction stage features a lot of RISC features
- MIPS has split I-Cache and D-Cache unlike VAX which has same I+D Cache
- MIPS has larger page size
- Same Cycle time
- MIPS has much faster MEM access ,FP ops

No Big Difference Now!

- Common Goal of High Performance will bring them together
  - Incorporating each other's features
  - Incorporating similar functional units.
    - Branch Prediction
    - OOE (Out of Order Executio) etc

An exception

- Embedded Processors
  - CISC is unsuitable
  - MIPS/watt ratio
  - Power consumption
  - Heat dissipation
  - Simple Hardware = integrated peripherals

From CISC to RISC (1)

- What Intel, the most famous CISC advocates, and HP do in IA-64:
  - Migrate to a Common Instruction Set.
  - Creating Small Instructions
  - More concise Instruction Set.
  - Shorter Pipeline
  - Lower Clock Cycle

From CISC to RISC (2)

- What Intel, the most famous CISC advocates, and HP do in IA-64:
  - Abandon the Out-of-order Execution In Hardware
  - Depend on Compiler to Handle Instruction Execution Order. Shifting the Complexity to Software.

From CISC to RISC (3)

- AMD Use Microcode and Direct Execution to Handle Control in Athlon

- CISC Datapaths Support Other RISC-like Features (such as register-to-register addressing and an expanded register count).

From RISC to CISC (1)

- Additional registers
- On-chip caches (which are clocked as fast as the processor)
- Additional functional units for superscalar execution

From RISC to CISC (2)

- Additional "non-RISC" (but fast) instructions
- On-chip support for floating-point operations
- Increased pipeline depth

CISC and RISC

- Incorporating Same Features
  - Complex Multi-level Cache
  - Branch Prediction
  - Out-of-order Execution

CISC vs RISC

- Hard to Distinguish Now. Boundary is getting vague.
- Academia don’t Care
- Industry doesn’t Care (Except for Advertisements)

RISC vs CISC

- Which one is better for general-purpose microprocessor design?
- It does not matter because
  - The main factor driving general-purpose microprocessor design has been the peculiar economics of semiconductor manufacturing

RISC vs CISC: 500k transistors

- For a few years in the late 80's, designers had a choice:
  - CISC CPU and no on-chip cache
  - RISC CPU and on-chip cache
- On-chip cache was probably a slightly better choice, giving RISC several years of modest advantage
- It is not RISC who gave better performance at this certain period; it was about the on-chip cache!

RISC vs CISC: 2M transistors

- Now possible to have both CISC and on-chip cache
- CISC can challenge RISC and it even has more advantage
- RISC chips become more CISC-like

Even More Transistors

- Then more transistors became available than single CISC CPU and reasonable cache could use… What now?
  - Multi-processor chips?
  - Superscalar?
  - VLIW?

Convergence: 5M transistors

- Superscalar won. But
  - It is really hard to pipeline and schedule superscalar computations when instruction cycles, word-lengths differ, and when there are 100s of different instructions
  - Compilers used only a small subset of instructions
- This pushed CISC designs to be more RISC-like

Even more: 50M transistors

- The economy of IC manufacturing have been making RISC and CISC go together
- Maybe one day these two become historic terms and ?ISC will prevail